LEVEL II

(12)

AD A104320

A TWO-STEP APPROACH TO FINITE ELEMENT ORDERING

by

Steven J. Fenves

Kincho H. Law

A Technical Report of
Research Sponsored by
THE OFFICE OF NAVAL RESEARCH
DEPARTMENT OF THE NAVY
Contract No. N00014-76-C-0354

August 1981

Department of Civil Engineering
Carnegie-Mellon University
Pittsburgh, Pennsylvania

R-81-130

DEPARTMENT OF CIVIL ENGINEERING
CARNEGIE INSTITUTE OF TECHNOLOGY
Carnegie-Mellon University

81 9 17 035

A TWO-STEP APPROACH TO FINITE ELEMENT ORDERING

by

Steven J. Fenves

Kincho H. Law

A Technical Report of
Research Sponsored by
THE OFFICE OF NAVAL RESEARCH
DEPARTMENT OF THE NAVY
Contract No. N00014-76-C-0354

Aug[illegible]981

Department of Civil Engineering
Carnegie-Mellon University
Pittsburgh, Pennsylvania

R-81-130

Table of Contents

LIST OF FIGURES

LIST OF TABLES

SUMMARY

A two-step approach to finite element ordering is introduced. The scheme involves ordering of the finite elements first, based on their adjacency, followed by a local numbering of the nodal variables. The ordering of the elements is performed by the Cuthill-McKee algorithm. This approach takes into consideration the underlying structure of the finite element mesh, and may be regarded as a "natural" finite element ordering scheme. The experimental results show that this two-step scheme is more efficient than the reverse Cuthill-McKee algorithm applied directly to the nodes, in terms of both execution time and the number of fill-in entries, particularly when higher order finite elements are used. In addition to its efficiency, the two-step approach increases modularity and flexibility in finite element programs, and possesses potential application to a number of finite element solution methods.

INTRODUCTION

The use of finite element methods typically involves solving a system of equilibrium equations :

$$K\ u = R \tag{1}$$

where u and R are, respectively, the displacement and the loading vectors. K is the global stiffness matrix which is often symmetric, positive definite, and populated with many zeros. In the solution process, it is important to take into account the structure of the matrix K, i.e. the pattern of the zero and nonzero entries so that the computational effort is minimized.

The solution process for solving Equation 1 can be divided into four separable tasks[12] :

1. Ordering - to find a proper permutation matrix P, such that the symmetric permuted matrix PKP^t has a desirable structure.
2. Storage allocation - to determine the necessary information about the structure of the matrix factor of PKP^t and to set up the storage scheme.
3. Numeric factorization - to decompose the permuted matrix PKP^t into a triple matrix product LDL^t.
4. Solution - to compute the solution vector u successively by a forward substitution, $z = L^{-1}(P^tR)$, and a backward substitution, $u = P(L^{-t}D^{-1}z)$.

The identification of these four separable tasks not only encourages software modularity, but also facilitates the theoretical study of sparse matrices. This paper focuses on the first task, that of ordering of a system of equilibrium equations resulting from finite element discretization.

When the matrix K is decomposed into its matrix factors L and D, it typically suffers some fill-in; that is, the filled matrix, $F = L + L^t$, has nonzeros in positions which are zero in K. One objective in ordering a matrix is to minimize the number of these fill-in entries. By doing so, the computing cost may also be reduced since the amount of computation depends on the number of nonzeros in the matrix factor L.

A finite element mesh consists of a collection of finite elements which are connected at their common boundaries. The discretization of the mesh and the finite elements are selected so that the behavior of the physical structure under examination can be properly modelled. The nodal variables are then defined on the finite element mesh; their number depends on the type of finite elements used. Traditionally, most ordering schemes have been developed for ordering directly the nodal variables and, hence, the equilibrium equations in 1. Very little effort has been reported in developing ordering schemes which include consideration of the underlying topological structure of the finite element mesh.

One solution scheme specifically designed for solving a system of equations resulting from finite element problems is the frontal solution method[14]. This method is often considered as the most "natural" solution scheme since it operates directly on the underlying structure of the finite element mesh. In this solution scheme, the finite elements are assembled and entered into the solution one at a time. A nodal variable is eliminated as soon as all the neighboring finite elements incident to it are available. The remaining assembled nodal variables, which are not yet ready for elimination, are retained in the "front", or "active front".

Another solution scheme which is similar to the frontal solution method is the so-called generalized element method[20]. In this method, the nodal variables of an active front are treated as a superelement. Multiple fronts are allowed, and each front is treated as a superelement. This method has recently attracted attention from numerical analysts[6]. The merit of the frontal solution and the generalized element methods is that both can be extended to include considerations of auxiliary storage in a simple manner.

Obviously, the efficiency of the frontal solution and the generalized element methods depends on the order in which the finite elements are numbered. The ordering of the nodal variables is implicitly imposed by the order in which the finite elements are processed. To the authors' knowledge, there have not been any studies reported for ordering the finite elements directly. In this paper, we attempt to examine the feasibility of ordering the finite elements.

It will be shown that ordering the finite elements has considerable computational advantage over ordering the nodal variables, but that ordering the finite elements alone does not completely define the permutation matrix P. It is therefore necessary to include into the ordering scheme a local ordering strategy for numbering the nodal variables of each finite element so that the number of fill-in entries can be minimized. This "two-step" approach, which includes the ordering of the finite elements as well as of the nodal variables, is the subject of this paper.

GRAPH THEORY NOTATION AND SPARSE MATRIX STUDY

The use of graph-theoretic approaches has found many applications in sparse matrix study. Although few results from graph theory are directly applicable to the study of sparse matrices, the graph representation is, nonetheless, a powerful tool to characterize the structure of a sparse matrix.

Symmetric Matrices and Finite Graphs

A finite undirected graph $G=(V,E)$ consists of a finite set V of n elements, called *nodes*, and a set E of unordered pairs of distinct nodes (v_i,v_j), called *edges*. For any node v in G, the set of nodes adjacent to v, adj(v), is defined as

$$\mathrm{adj}(v) = \{ v_i \in V \mid (v,v_i) \in E \}.$$

The number of nodes in adj(v), denoted by $|\mathrm{adj}(v)|$, is called the *degree* of v. The *deficiency* of v, def(v), is the set of distinct pairs of nodes in adj(v) which are not themselves adjacent. A graph is *complete* if every pair of nodes is adjacent. A *subgraph* $G'=(V',E')$ of G is a graph for which $V' \subseteq V$ and $E' \subseteq E$. A complete subgraph is called a *clique*. A *section graph* $G(V')$ is a subgraph $G=(V',E(V'))$ induced by a node set V', where

$E(V') = \{ (v_i,v_j) \in E \mid v_i,v_j \in V' \}$

If a set of nodes V" is deleted from the graph G=(V,E), the section graph G(V\V") is obtained from G by removing the nodes V" together with their incident edges.

A (simple) *path* $\{v_i, ..., v_j\}$ is a sequence of distinct nodes and continuous edges leading from v_i to v_j such that there are no repeating edges. A graph G is said to be *connected* if there is at least one path between every pair of distinct nodes in G; otherwise, G is disconnected. A connected subgraph is called a *component*. The *distance* $d(v_i,v_j)$ between two nodes v_i and v_j in a connected graph is the number of edges in the shortest path joining nodes v_i and v_j. The *eccentricity* e(v) of a node v is then given by

$e(v) = \max \{ d(v,v_i) \mid v_i \in V \}$.

The *diameter* $\delta(G)$ of the graph G=(V,E) is defined as

$\delta(G) = \max \{ e(v), v \in V \}$.

A node v is said to be a *peripheral node* if $e(v) = \delta(G)$.

For a graph G=(V,E) with n nodes, an *ordering* (or *labelling*) of V is a bijection (a one-to-one onto mapping) :

$\alpha : \{ 1, 2, ..., n \} \longleftrightarrow V$.

The ordered graph of G is denoted by $G_\alpha = (V,E,\alpha)$. The integer, ranging from 1 to n, assigned to a node by the ordering is called the *number* or *label* of that node.

The notation of a *level structure*[2] is commonly used to describe the properties of many ordering schemes. A level structure LS of a graph G is an arrangement of the nodes of G into m levels, ordered $LS_1, ..., LS_m$, such that nodes at a given level LS_k are connected to no nodes other than LS_{k-1}, LS_k and LS_{k+1}. For a node $v \in V$, there corresponds a level structure *rooted* at v. The levels of the rooted level structure are determined by

1. assigning $LS_1 = \{v\}$, and
2. for $k > 1$, assigning to LS_k the set of nodes adjacent to nodes in LS_{k-1} which have not yet been included in any previous levels.

Given an n by n symmetric matrix K, there corresponds to it a finite graph G=(V,E), where a node $v_i \in V$ denotes the ith row (or column) of the matrix K, and an edge $(v_i,v_j) \in E$

symbolizes an offdiagonal nonzero entry K_{ij} ($= K_{ji}$). An ordering of the graph G of matrix K is equivalent to a symmetric permutation PKP^t, where P is an n by n permutation matrix. The unordered graphs (in which nodes are not labelled) of K and PKP^t are the same, but the node numbers of the associated ordered graphs are different.

Finite Element Mesh and Its Solution Graph

A finite element mesh is a collection of finite elements in which the adjacent finite elements are joined at their common boundaries or vertices. There is a node at each vertex of the finite element mesh. For finite elements of higher order, where higher order polynomial interpolation functions are used, there may also be nodes lying along the sides or on the faces of the finite elements, and/or located internally within the finite element itself. The nodes situated at the interior of the finite element are referred as the *internal variables*, and the nodes located along the boundaries are called the *external variables*.

Associated with each node is a set of variables corresponding to the degrees of freedom defined at that node. This set of variables, in turn, corresponds to a submatrix in the global stiffness matrix, which is typically full. During assembly and solution, this submatrix can conveniently be treated as a single entity. For simplicity, the set of variables at a node is referred as the *nodal variable*.

The finite element mesh can be transformed directly into the finite graph G representing the structure of the global stiffness matrix. We call this graph a *finite element solution graph*, or, simply, a *solution graph* (to distinguish it from the finite element connectivity graph to be introduced below). The nodes of the solution graph are the nodal variables defined on the finite element mesh. The edges of the solution graph are constructed by making the nodes of each finite element pairwise adjacent, since a nonzero entry K_{ij} in the global stiffness matrix K implies that the i^{th} and the j^{th} nodal variables share at least one incident finite element. In other words, if V' denotes the set of nodal variables belonging to one finite element, the section graph G(V') of the solution graph G is a clique; the union of the cliques over all elements defines the edges in the solution graph. Examples of transforming a finite element mesh into its associated solution graph are shown in Figure 1.

Matrix Factorization and Its Graph Theoretic Model

The factorization of K can be symbolically modelled using a graph-theoretic approach. This approach is particularly helpful in understanding how the fill-in entries are created during factorization.

The most fundamental scheme of matrix factorization is one analogous to Gaussian elimination, summarized in Figure 2. At the i^{th} step of the factorization, the i^{th} column of L and the i^{th} diagonal entry of D are computed, and the matrix $K^{(i)}$ is condensed and modified to $K^{(i+1)}$. It is in the condensation that the fill-in entries, if any, are created.

Using the finite graph representation of a sparse matrix, the factorization scheme can be modelled graph-theoretically as a node-elimination process[18]. Upon elimination of node v, the elimination graph G_v is obtained from $G=(V,E)$ by :

1. deleting node v and its incident edges;
2. adding auxiliary (fill-in) edges such that all adjacent nodes of v form a clique.

That is,

$$G_v = (V\backslash v,\ E(V\backslash v)\ U\ def(v)).$$

For an ordered graph G_α, the elimination is a recursive process defined by

$$P(G_\alpha) = \{\ G_0 = G_\alpha,\ G_1,\ \ldots,\ G_{n-1}\ \}$$

where G_i is called the i^{th} elimination graph obtained by eliminating v_i from G_{i-1}. The fill-in edges created during the elimination of v_i, denoted by f_i, are the $def(v_i)$ in G_{i-1}. These fill-in edges correspond to the new nonzero entries introduced into the condensed matrix $K^{(i+1)}$ at the i^{th} step of the factorization.

In a finite element solution graph, the new clique formed by the elimination of a node can be treated as a new finite element, called a generalized element or a superelement[20]. The node-elimination process on a finite element graph can thus be interpreted as a series of transformation of the finite elements into the superelements. This elimination model also suggests an interesting characteristic related to higher order finite elements. Internal nodes of a higher order element are connected only to the external nodes of that element. Furthermore, all nodes in a finite element form a clique. Hence, if an internal node is eliminated before any of the external nodes of the same element, there is no fill-in since the deficiency of the internal node is null.

After elimination, the set of fill-in edges is given by

$$\Phi(G_a) = \bigcup_{i=1}^{n-1} f_i .$$

The filled graph G_F, which represents the structure of the filled matrix, $F = L + L^t$, is defined by

$$G_F = (V, E \cup \Phi(G_a)).$$

The node-elimination model was formally introduced by Rose[18]. Since then, efforts have been made to develop efficient algorithms implementing the model[19, 11]. These algorithms have been encoded into sparse matrix programs such as YMSL[7] and Sparspak[9]. The major application of these algorithms is in the second task presented in the Introduction, namely to set up the data structure for storing the numeric entries of the matrix factors.

A node-addition model which simulates the Cholesky factorization has recently been developed by the authors[15]. For this graph-theoretic model, nodes are added onto the filled graph G_F, rather than being eliminated from the original graph G_a. The structure of the matrix factor L is constructed one row at a time. When computing the entries of a particular row in L, the model does not require any a priori information for the rows beyond the current row. Therefore, this model has the flexibility that the tasks of labelling a nodal variable, determining its row structure in L, and computing the numeric entries of the row can all be performed simultaneously.

The execution times for the numeric factorization as well as the symbolic factorization of a given matrix K depend on the number of nonzeros in the matrix factor L. Therefore, it is worthwhile to develop efficient algorithms to order the matrix K so that the number of fill-in entries, or equivalently the number of nonzeros in L, is minimized. In the next section, two well known ordering algorithms, namely the Cuthill-McKee algorithm and the reverse Cuthill-McKee algorithm, are discussed.

The Cuthill-McKee and the Reverse Cuthill-McKee Ordering Algorithms

Let K be an n by n symmetric matrix. For the ith row of K, define

$$\sigma_i(K) = \min \{ j \mid K_{ij} \neq 0 \}$$

and

$$\beta_i(K) = i - \sigma_i(K).$$

The number $\sigma_i(K)$ denotes the column subscript of the first nonzero entry in the ith row of matrix K. The number $\beta_i(K)$ is usually referred as the *local bandwidth* of the ith row of matrix K, and is equal to the number of off-diagonal entries between the first nonzero of the row and the main diagonal. The *bandwidth* and the *profile* of matrix K are defined as

$$b(K) = \max \{ \beta_i(K), i = 1, ..,n \}$$

and

$$\rho(K) = \sum_{i=1}^{n} \beta_i(K)$$

respectively. The nonzero entries of the filled matrix F of K are confined within the local band of each row. Many ordering schemes have been developed to minimize the bandwidth or the profile of K.

One ordering scheme commonly used with finite element programs to minimize the bandwidth is the Cuthill-McKee algorithm[4]. Basically, the algorithm is a breadth-first technique in labelling the nodes of a graph. Once a finite element mesh is tranformed into its finite element solution graph, the algorithm numbers the nodes in the following way.

1. Determine a starting node and number it v_1.
2. For nodes v_i, $i=1,...,n$, find all unlabelled neighboring nodes of the node v_i and number them sequentially in increasing order of degree.

This ordering algorithm is essentially the same as that for generating a level structure rooted at the starting node v_1.

In selecting a starting node, a peripheral node is preferred[13]. The idea of choosing a peripheral node as a starting node is to generate as many levels as possible in the level structure, since an increase in the number of levels tends to decrease the profile of the corresponding matrix. Unfortunately, existing algorithms for finding a peripheral node are not computationally feasible.

A compromise that has been suggested is to choose a starting node with high eccentricity. This node is generally referred as a *pseudo-peripheral node*. An efficient algorithm to find a pseudo-peripheral node has been suggested and implemented by George and Liu[10]. For most practical cases, the execution time for finding a pseudo-peripheral node is no greater than $O(|E|)$, i.e., the order of the number of edges in the graph $G=(V,E)$.

George et.al.[12] have shown that if linear insertion is used for sorting the time complexity for the second step of the Cuthill-McKee algorithm requires at most

$(4|E| + 2cm|E|)$ operations,

where $|E|$ is the number of edges in the graph G, m is the maximum degree of any node, and c is some constant.

It has been discovered that significant improvements in minimizing the profile can be achieved by simply reversing the node ordering obtained from the Cuthill-McKee algorithm[8]. The resulting algorithm is the well known reverse Cuthill-McKee (RCM) algorithm, which can be summarized as follows :

1. number the nodes by the Cuthill-McKee algorithm; and
2. reorder the nodes by reversing the node numbering obtained in Step 1.

To reverse the ordering of n nodes requires only n operations. Therefore, the overall complexity of the RCM algorithm remains bounded by $O(m|E|)$.

It has been proved by Liu and Sherman that the RCM algorithm is never inferior to the Cuthill-McKee algorithm[16]. In its application to finite element ordering, they also found that the RCM algorithm is particularly superior when finite elements of higher orders are used.

A listing of computer subroutines for the RCM algorithm can be found in reference 12. A more detailed description of the algorithm can also be found in that reference.

A TWO-STEP APPROACH TO FINITE ELEMENT ORDERING

In this section, we present a "two-step" approach to finite element ordering. The ordering process is divided into two separate tasks. The first task orders the finite elements in the finite element mesh. The second task then labels the nodal variables.

Representation of Finite Element Connectivity

The method to be described requires an explicit representation of the connectivity of the finite elements in the mesh. The connectivity of the finite elements can be topologically represented as a graph; we call this graph a *finite element connectivity graph*, or, simply, a *connectivity graph*. The nodes in the connectivity graph correspond to the finite elements in the mesh. The edges of the connectivity graph are used to describe the interconnections between the finite elements. One possible way to define the interconnections between the finite elements is to say that two finite elements are adjacent if they share a common node in the mesh. This definition, however, may lead to a huge number of edges in the connectivity graph. Since the efficiency of most existing ordering algorithms is a function of the number of edges in the graph, this representation of finite element connectivity may suffer a significant drawback in terms of the execution time required for ordering the finite elements. Here, we examine an alternative definition in which the number of edges in the finite element connectivity graph can be vastly reduced.

In finite element methods, a continuum is subdivided by imaginary lines or surfaces into a number of finite elements. These elements are assumed to be interconnected at the vertices situated on their boundaries. Therefore, it is simple to assert that the finite elements are topologically interconnected by their boundaries. A finite element of n dimensions, where n = 1,2,3, posseses boundaries of (n-1) dimensions. For instance, the boundaries of a 3-dimensional (volumetric) finite element are the 2-dimensional boundary-faces, so that in a 3-dimensional continuum, the volumetric finite elements are interconnected by their boundary-faces. In the same fashion, 2-dimensional (planar) finite elements are bounded and interconnected by their 1-dimensional boundary-lines, and 1-dimensional (linear) finite elements are connected to their adjacent elements through the 0-dimensional boundary-nodes. Hence, in a finite element mesh, two finite elements are said to be adjacent if they are connected at their boundaries, rather than at their vertices. Using this definition, the nodes in the finite element connectivity graph are the finite elements in the finite element mesh, while the edges connecting the nodes in the connectivity graph correspond to the imaginary boundaries of the finite elements separating the continuum. Examples of finite element meshes and their associated connectivity graphs are shown in Figure 3. For a two-dimensional (planar) continuum, it is interesting to note that the finite element connectivity graph is analogous to the dual of a planar graph[5], with the exterior node in the dual graph omitted. In general, the element-element connectivity relationship is topologically equivalent to the definition of a dual of an n-dimensional complex[23, 3].

In some cases, the connectivity of finite elements cannot be completely represented using the definition given above. As shown in Figure 4(a), n-dimensional finite elements may not necessarily be connected through all their (n-1)-dimensional boundaries. Another example may be that the adjacent finite elements do not have the same geometric dimensions, as illustrated in Figure 4(b). Nevertheless, the transformation process described is still valid and applicable to these cases; the resulting finite element connectivity graph may at worst become a disconnected graph. Each connected component in the connectivity graph can be labelled independently by the method presented. The nodes at the interface between two connected components must then be numbered higher than all other nodes in the two components. The examples just described are not very common in practice, and will not be pursued further.

There are several major advantages in defining the connectivity of the finite elements by means of their boundaries, instead of their nodes. First, the number of edges in the connectivity graph is minimized. Furthermore, this definition completely disregards the nodal variables, or vertices, in the finite element mesh. Therefore, the ordering of finite elements can be performed before the types of finite elements, and thus the number of nodal variables, are chosen.

Ordering the Finite Elements

Once the adjacency structure of a connectivity graph has been established, the Cuthill-McKee algorithm can be applied to number the nodes of the connectivity graph, which correspond to the finite elements in the mesh. An example of a 5 by 5 regular mesh with triangular finite elements is shown in Figure 5(a). The mesh is first transformed into its connectivity graph shown in Figure 5(b). The nodes in the connectivity graph are ordered by the Cuthill-McKee algorithm. The resulting ordering of the finite elements is given in Figure 5(c).

As noted earlier, the time complexity of the Cuthill-McKee algorithm is a function of the number of edges in a graph. For a given finite element mesh, the number of edges in the associated connectivity graph is considerably less than the number of edges in its finite element solution graph defined previously. When higher order finite elements are used, the difference is even more dramatic, since the number of edges in a finite element solution graph increases combinatorially with the number of nodal variables per finite element. On the other hand, the number of edges in the finite element connectivity graph remains constant for a given discretization of the finite element mesh.

Let the finite element mesh consist of ne finite elements, and let nb_i denote the number of boundaries of finite element i. The overall time complexity of the Cuthill-McKee algorithm, following from the previous discussion, is $O(m|E|)$, where m is the largest value of nb_i, $i = 1, \ldots, ne$, and $|E|$ is the total number of internal boundaries interconnecting the finite elements.

Ordering the Nodal Variables

A good ordering of the finite elements does not automatically guarantee a good node ordering, in the sense that the number of fill-in entries is minimized. The ordering of the finite elements does not complete the ordering of K, as the nodal variables associated with the element do not acquire labels by this process. Therefore, one must also consider the ordering of the nodal variables. The proposed strategy is to label the nodal variables of the finite elements following the order in which the finite elements are numbered. For each finite element, the nodal variables are ordered according to their valencies, where the *valency* of a node is the number of finite elements incident at that node. The nodal valency is an indicator of the degree of a node.

The local ordering scheme to label the nodal variables is summarized as follows :

1. Determine the nodal valency for each node in the finite element mesh.
2. (Main loop) Enter the finite elements one at a time following the order in which the finite elements were previously numbered by applying the Cuthill-McKee algorithm to the connectivity graph. For each finite element, find all unlabelled nodes connected to it and number them sequentially in descreasing order of their valencies.
3. (Reverse ordering) Reverse the ordering of the nodal variables obtained in Step 2.

In Step 2 of the local ordering scheme, a nodal variable with minimum valency among the unlabelled nodal variables in a finite element is numbered last. The node ordering obtained is then reversed in Step 3. The motivation behind these two steps is that the nodal variables with the lowest valency in a finite element will be eliminated first. This strategy of minimum degree (valency) ordering has been employed in many popular node ordering schemes, such as the minimum degree ordering algorithm[18] and the Cuthill-McKee algorithm[4], although the strategy is used in a different manner in different schemes.

It has been mentioned that by reversing the node ordering obtained from the Cuthill-McKee algorithm, the result can be improved significantly. This property is also true in the local ordering scheme proposed. First, reversing the numbering of the nodal variables which are ordered in decreasing order of their valencies ensures that internal nodal variables are numbered before the external variables of the same element, so that the internal nodal variables are eliminated before the external nodal variables of the same finite element. Hence, there will be no fill-in created when eliminating the internal nodal variables. This ordering strategy also has the property that the nodes situated along a boundary are numbered and eliminated before the corner nodes of the same boundary. This is because the corner nodes are in general connected to more finite elements than the nodes located along a boundary. The nodal numbering produced by the proposed scheme for a simple example of a 5 by 5 regular mesh consisting of 10-node triangular finite elements is shown in Figure 6. The two properties just described are clearly demonstrated by this example.

For step 1, given the element-node incidence table -- a listing of the nodes incident on the finite elements -- the determination of the nodal valencies can be done in exactly

$$|T| = \sum_{i=1}^{ne} nv_i \quad \text{operations,}$$

where nv_i is the number of nodal variables of the i^{th} finite element, ne is the number of finite elements in the mesh, and $|T|$ is the size of element-node incidence table.

To implement step 2, a linear insertion may be used to sort the unlabelled variables of each finite element. For some constant c, the sorting of nv_i elements by linear insertion requires $c(nv_i^2)$ operations[1]. Thus, the time complexity of Step 2 of the algorithm requires at worst

$$c(\sum_{i=1}^{ne} nv_i^2) \leq c(\sum_{i=1}^{ne} nv_i)\, nv_{max} = c|T|nv_{max} \quad \text{operations,}$$

where $nv_{max} = \max(nv_i)$, $i=1,\ldots,ne$. The number of operations required to reverse the node ordering equals to the number of nodal variables, nn, in the finite element mesh. Hence, the time complexity for numbering the nodes, given the ordering of the finite elements, is bounded by $(O(|T|nv_{max}) + nn)$.

EXPERIMENTAL RESULTS

In this section, we report some experimental results comparing the reverse Cuthill-McKee algorithm and the two-step ordering algorithm. Four finite element models have been selected as examples. They are : (1) a 5 by 5 regular mesh with triangular finite elements; (2) an L-shaped mesh with triangular finite elements; (3) a cross-shaped mesh with rectangular finite elements; and (4) a 2 by 2 by 2 model consisting of rectangular block finite elements. These models are shown in Figure 7. For each model, linear, quadratic and cubic finite elements have been used. There are altogether twelve test cases.

The computer subroutines for the RCM algorithm have been adopted from the Sparspak package developed by George and Liu at the University of Waterloo[9] and listed in Reference 12. These subroutines include finding a pseudo-peripheral node of a graph and ordering the nodes by the

RCM algorithm. The input information to these subroutines is the number of nodal variables and an adjacency structure pair representing the node connectivity of the solution graph. A node adjacency structure is illustrated in Figure 8(a).

For the two-step ordering scheme, the same set of computer subroutines from Sparspak has been employed to number the finite elements by the Cuthill-McKee algorithm, except that the step for reverse ordering has been omitted. In this case, the input data for the adjacency structure pair is the node connectivity of the finite element connectivity graph. An example of this data structure is given in Figure 8(b). Once the finite elements have been ordered, the adjacency structure can be discarded. The subroutine for performing the second step, that of ordering the nodal variables, is listed in Appendix I. For this subroutine, the element-node incidence table is assumed to be available.

The test cases have been run on a DECsystem 20 computer at Carnegie-Mellon University. For each test case, the execution times for the two-step ordering scheme and the RCM algorithm are reported in Table 1. To minimize timing errors due to a multi-programmed operating system environment, the test cases have been run when the computer was lightly loaded. As the results indicate, the two-step ordering scheme requires much less execution time than the RCM node ordering algorithm, except for the cases when linear triangular finite elements are employed. For those cases, the execution times for ordering the nodes by the RCM algorithm and for ordering the finite elements by the Cuthill-McKee algorithm are approximately the same. The deficiency of the two-step ordering scheme is due to the second step of labelling the nodal variables. For cases when higher order finite elements are used, very large amounts of savings in execution times can be achieved.

For each test case, the structures of the matrix factor L resulting from the RCM and the two-step ordering algorithms are summarized in Table 2, in terms of the profile of the stiffness matrix, the number of fill-in entries and the size of the matrix factor. For almost all cases, the profiles resulting from the two-step ordering scheme are slightly larger than those obtained from the RCM algorithm. On the other hand, the two-step ordering scheme, in most cases, leads to less fill-in than the RCM algorithm. This result is particularly true for finite element meshes where higher order finite elements are used.

DISCUSSION

In this paper, a "two-step" finite element ordering scheme has been introduced. In addition to the efficiency achieved, the scheme is also highly adaptable to various solution methods commonly used in finite element analysis.

The major characteristic of the two-step ordering scheme is that ordering the finite elements and ordering the nodal variables are separated into two independent tasks. The two tasks can be implemented as two separate modules. This property of modularity provides flexibility in the software design of finite element programs. For instance, one can choose to number the finite elements once the discretization of the finite element mesh is established, even without the knowledge about the types of finite elements to be used. In view of current software developments in generating finite element meshes using graphic preprocesssors, the task of numbering the finite elements can easily be incorporated as an additional module in the mesh generator with very little extra cost.

The authors recognize that the two-step ordering scheme presented requires two sets of input data: one to describe the adjacency of the finite elements and the second for the element-node incidence table. These two sets of data can both be generated by a mesh generator. Conventionally, however, only the element-node incidence table is created. In Appendix II, we present a computer program to determine the element-element adjacency structure pair using the element-node incidence table as input. This computer program serves primarily for illustration purposes, and is not meant to be efficient. The authors emphasize that the finite element adjacency structure should be generated by the mesh generator, particularly when a graphical preprocessor is used.

Since the birth of the finite element methods, efforts have been made to develop finite elements using polynomial interpolation functions of higher orders. The experimental results favor strongly the use of the two-step ordering scheme with higher order finite elements. In many finite element problems, the finite elements are progressively modified by using interpolation functions of increasingly higher order so as to improve the accuracy of the solution. At each modification, the structure of the global stiffness matrix changes, but the number and the distribution of the finite elements in the mesh often remain unchanged. Examples of these problems are quality control in finite element analysis and nonlinear structural analysis[21]. For this type of problem, the ordering of finite elements needs only be determined once. The result can be used to reorder the nodal variables upon each modification made to the finite elements. Moreover, the step in ordering the nodal variables by the two-step ordering scheme can be performed much faster than a complete re-numbering by the node ordering schemes.

In large scale structural analysis, substructuring is often used to divide the structure into two or more smaller components, called substructures, which are interconnected at their boundaries. Topologically, the substructures can be treated in the same manner as finite elements. Therefore, the strategy for ordering the finite elements may well be applied to the ordering of the substructures. In substructuring analysis, the external nodal variables located along the boundaries of the substructures are ordered last. As discussed previously, this characteristic is also embedded in the node ordering strategy of the two-step scheme. Hence, there is no reason that the two-step approach cannot be extended to an ordering scheme for substructuring methods.

One of the characteristics of the frontal solution method is that the assembling of the finite elements and the solution process are performed simultaneously. For this solution method, it is the numbering of the finite elements that really matters. However, a proper ordering of the nodal variables for each finite element can further reduce the number of fill-in entries and, thus, the computation cost of the solution process. In the two-step scheme presented, the strategy is to label the nodal variables following the ordering of the finite elements; then the final node ordering is reversed. As a result, the order in which the nodal variables are to be eliminated ought to follow the reverse order in which the finite elements are numbered. Therefore, by entering the finite elements in the reverse order of their numberings, the assembling and factorization tasks can also be performed simultaneously. However, unlike the frontal solution method, where the nodal variables are arranged in an arbitrary order, the nodal variables are pre-ordered by the two-step ordering scheme to reduce the number of fill-in entries. This preordering of the nodal variables has the advantage that the data structure for the matrix factors can be set up independently. *Throughout the entire ordering scheme, the structure of the global stiffness matrix need not exist.* With the nodal variables pre-ordered, one can proceed directly to construct the data structure for the matrix factors using existing symbolic factorization algorithms [9, 15]. Consequently, while this two-step ordering scheme can improve the software modularity for finite element programs, many characteristics of the "natural" frontal solution method can still be retained in the solution process.

The authors do not claim that the two-step ordering scheme proposed in this paper is optimal in minimizing the number of fill-in entries. In fact, it has recently been proved that the problem of computing the minimum fill-in is NP-complete[24]. This ordering scheme is recommended because of its efficiency, modularity and flexibility, and its potential application to various other finite element problems. Most of all, the two-step ordering scheme includes the consideration of the underlying topological structure of the finite element mesh, and may therefore be regarded as a "natural" finite element ordering scheme.

ACKNOWLEDGEMENT

This work was supported by the Office of Naval Research under Grant N00014-76-C-0354.

REFERENCES

1. Aho. A.V., J.E. Hopcroft and J.D. Ullmann. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

2. Arany, I., and L. Szoda. An Improved Method for Reducing the Bandwidth of Sparse Symmetric Matrices. Information Processing 71, Proceedings of IFIP Congress 71, IFIP, August, 1971, pp. 1246-1250.

3. Branin, F.H. Jr. The Algebraic-Topological Basis for Network Analogies and the Vector Calculus. Tech. Rept. TR 00.1495, Systems Development Division, IBM, July, 1966.

4. Cuthill, E., and J. McKee. Reducing the Bandwidth of Sparse Symmetric Matrices. Proc. 24th Nat. Conf. of the ACM, Association for Computer Machinery, 1969, pp. 157-172.

5. Deo, N.. *Graph Theory with Applications to Engineering and Computer Science*. Prentice-Hall, Inc., 1974.

6. Duff, I.S. Recent Developments in the Solution of Large Sparse Linear Equations. In *Computing Methods in Applied Sciences and Engineering*, R. Glowinski and J.L. Lions, Eds., North Holland Publishing Company, 1980, pp. 407-426.

7. Eisenstat, S.C., M.C. Gursky, M.H. Schultz, and A.H. Sherman. Yale Sparse Matrix Package I. The Symmetric Codes. Research Report 112, Dept. Of Comp. Sci., Yale University, July, 1977.

8. George, A. Computer Implementation of the Finite Element Method. Tech. STAN-CS-71-208, Computer Science Dept., Stanford University, 1971.

9. George A., and J.W.H. Liu. User Guide for Sparspak: Waterloo Sparse Linear Equations Package. Research Report CS-78-30, University of Waterloo, 1978.

10. George, A., and J.W.H. Liu. "An Implementation of a Pseudo-Peripheral Node Finder." *ACM Trans. on Math. Software 5* (1979), 139-162.

11. George, A., and J.W.H. Liu. "An Optimal Algorithm for Symbolic Factorization of Symmetric Matrices." *SIAM J. Comput. 9*, 3 (August 1980), 583-593.

12. George, A., and J.W.H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall Inc., 1981.

13. Gibbs, N.E., W.G. Poole Jr., and P.K. Stockmeyer. "An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix." *SIAM J. Numer. Anal. 13*, 2 (April 1976).

14. Irons, B.M. "A Frontal Solution Program for Finite Element Analysis." *International Journal for Numerical Methods in Engineering 2* (1970), 5-32.

15. Law, K.H., and S.J. Fenves. Sparse Matrices, Graph Theory, and Reanalysis. Proceedings of the First International Conference on Computing in Civil Engineering, American Society of Civil Engineers, 1981.

16. Liu, W-H., and A.H. Sherman. "Comparative Analysis of the Cuthill-McKee and the Reverse Cuthill-McKee Ordering Algorithms for Sparse Matrices." *SIAM J. Numer. Anal. 13*, 2 (April 1976), 198-213.

17. Maurer, W.D., and T.G. Lewis. "Hash Table Methods." *ACM Computing Surveys 7*, 1 (March 1975), 5-19.

18. Rose, D.J. A Graph-Theoretic Study of the Numerical Solution of Sparse Positive Definite Systems of Linear Equations. In *Graph Theory and Computing*, R.C. Read, Ed., Academic Press, 1972, pp. 183-217.

19. Rose, D.J., R.E. Tarjan, and G.E. Lueker. "Algorithmic Aspects of Vertex Elimination on Graphs." *SIAM J. Comput. 5*, 2 (June 1976), 266-283.

20. Speelpenning, B. The Generalized Element Method. Tech. Rept. UIUCDCS-R-78-946, Dept. of Computer Science, University of Illinois at Urbana-Champaign, November, 1978.

21. Szabo, B.A., P.K. Basu and D.A. Dunavant. Quality Control in Finite Element Analysis. Proceedings of the First International Conference on Computing in Civil Engineering, American Society of Civil Engineers, 1981.

22. Tarjan, R.E., and A.C.C. Yao. "Storing a Sparse Table." *Communications of the ACM 22*, 11 (November 1979), 606-611.

23. Veblen, O.. *Colloquium Publications*. Volume V, Part II, Second Edition: *Analysis Situs*. American Mathematical Society, 1931.

24. Yannakakis, M. "Computing the Minimum Fill-in is NP-Complete." *SIAM J. Alg. Disc. Meth. 2*, 1 (March 1981), 77-79.

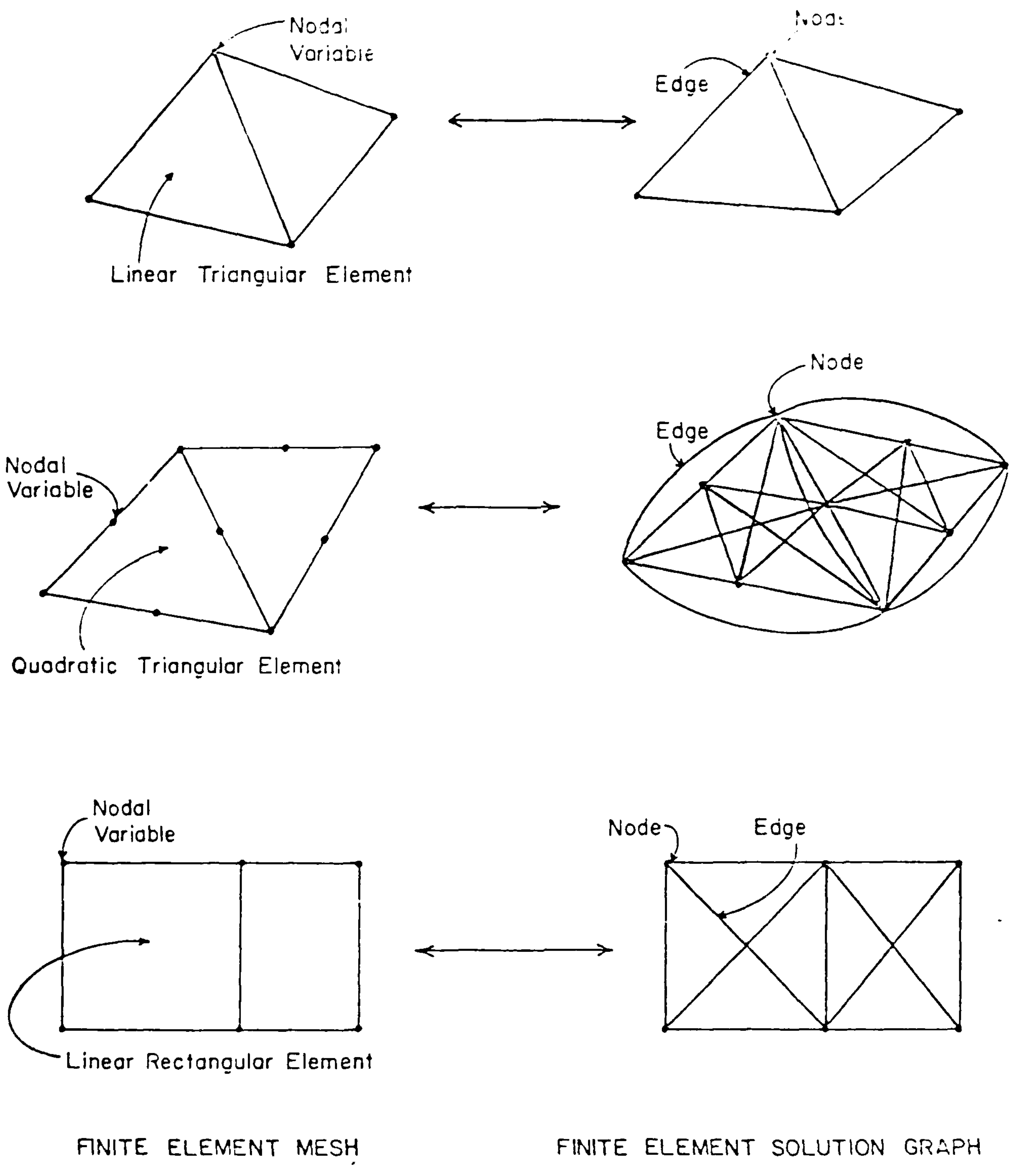
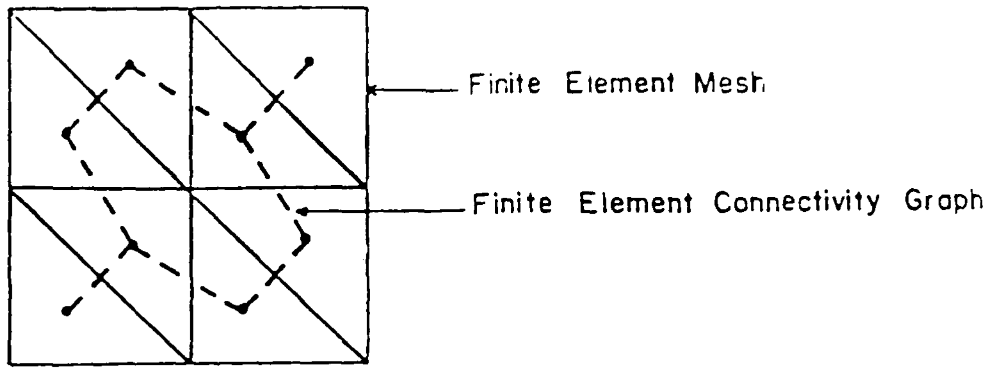

Nodal Variable
Linear Triangular Element
Node
Edge
Nodal Variable
Quadratic Triangular Element
Node
Edge
Nodal Variable
Linear Rectangular Element
Node
Edge
FINITE ELEMENT MESH
FINITE ELEMENT SOLUTION GRAPH

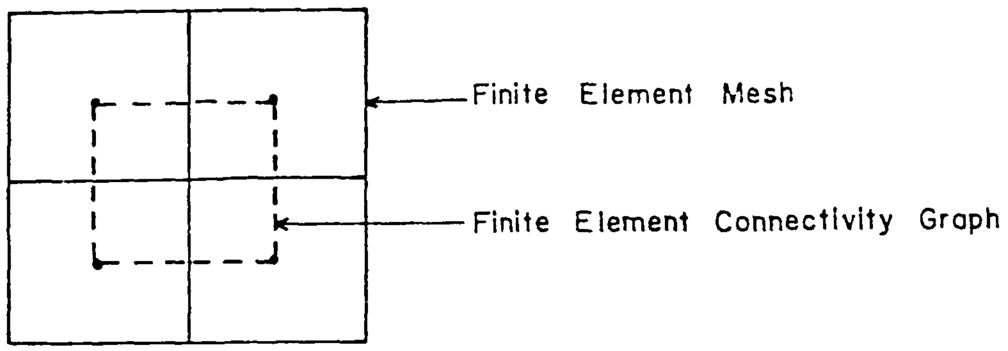
Figure 1: Finite Element Mesh and Its Associated
Finite Element Solution Graph

```
Algorithm : Matrix factorization
COMMENT : { Factoring K into $LDL^t$ }
BEGIN
    Set $K^{(1)} = K$;
    FOR column i = 1 UNTIL n; DO
    BEGIN
        $D_{ii} = K_{ii}^{(i)}$;
        FOR row j = i+1 UNTIL n; DO
        BEGIN
            $L_{ji} = K_{ji}^{(i)} D_{ii}^{-1}$;
            FOR column k = i+1 UNTIL j; DO
            BEGIN
                $K_{jk}^{(i+1)} = K_{jk}^{(i)} - L_{ji} D_{ii} L_{ki}^{t}$;
            END; (* for column k *)
        END; (* for row j *)
    END; (* for column i *)
END.
```

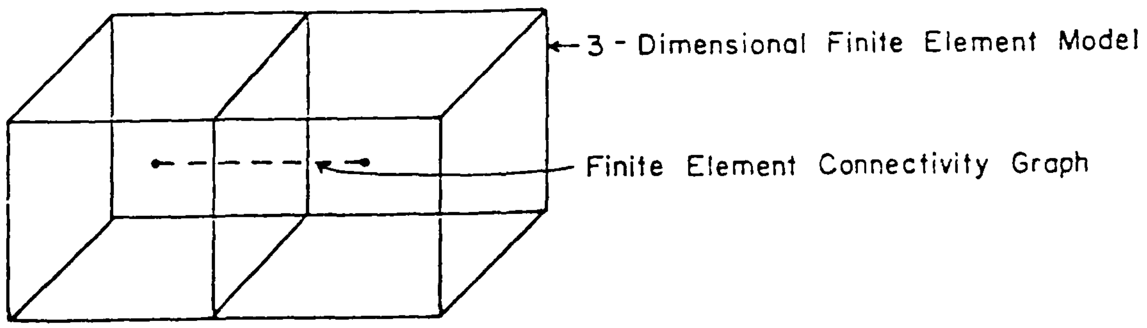
Figure 2. Matrix Factorization



(a) Finite element mesh with triangular elements

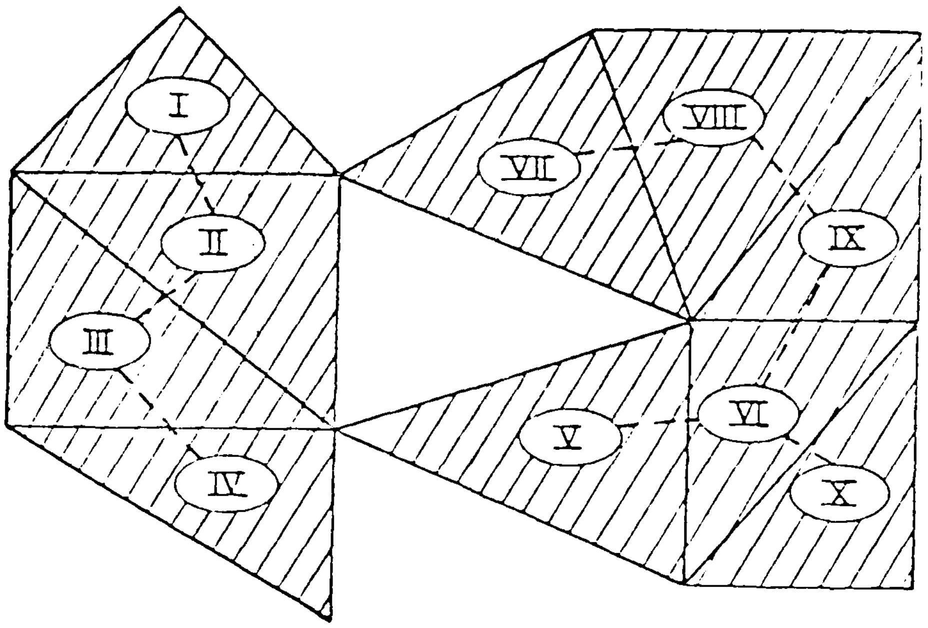


(b) Finite element mesh with rectangular elements

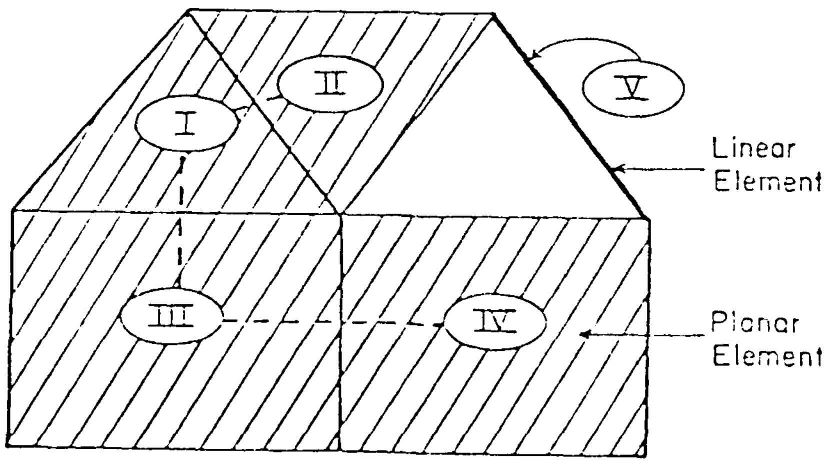

3 - Dimensional Finite Element Model
Finite Element Connectivity Graph

(c) 3-dimensional finite element model with rectangular volumetric elements

Figure 3: Finite Element Model and Its Assoicated Finite Element Connectivity Graph

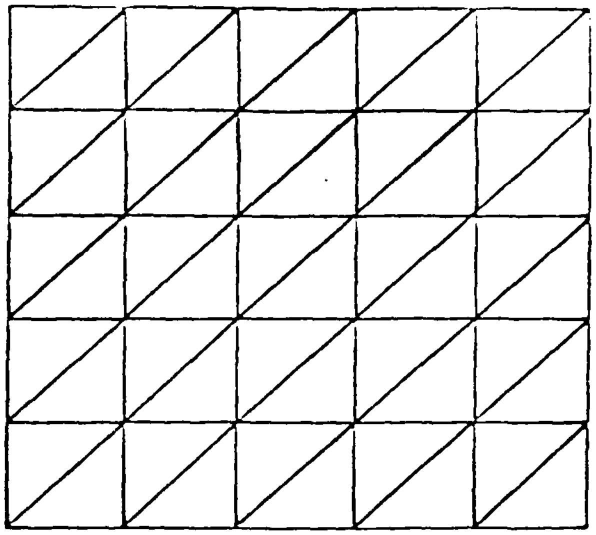


(a) Finite element mesh with n dimensional elements connected through boundaries other than those of (n-1) boundaries

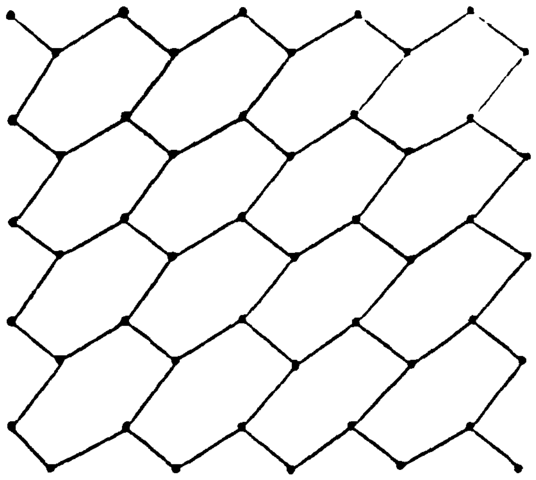


(b) Mesh with finite elements of different dimensions

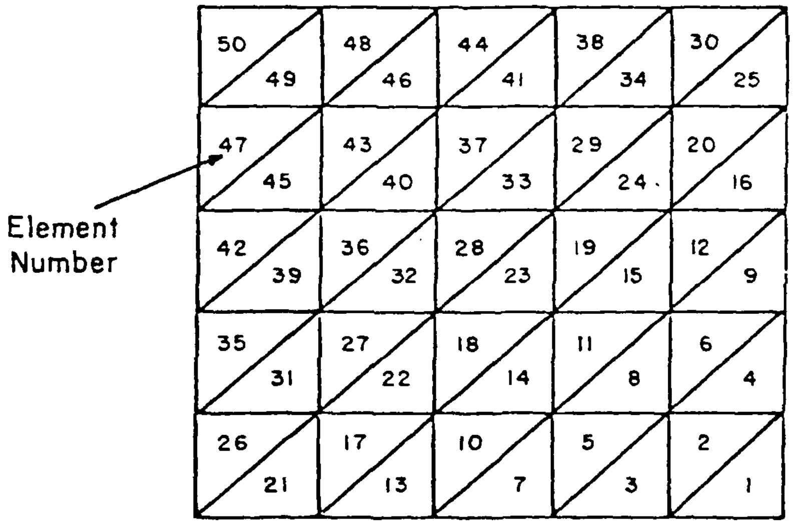
Figure 4: Finite Element Models with Uncommon Connectivities



(a) A 5 by 5 finite element grid



(b) Connectivity graph of the 5 by 5 finite element grid



(c) The ordering of finite elements

Figure 5: Ordering the Finite Elements by the Cuthill-McKee Algorithm

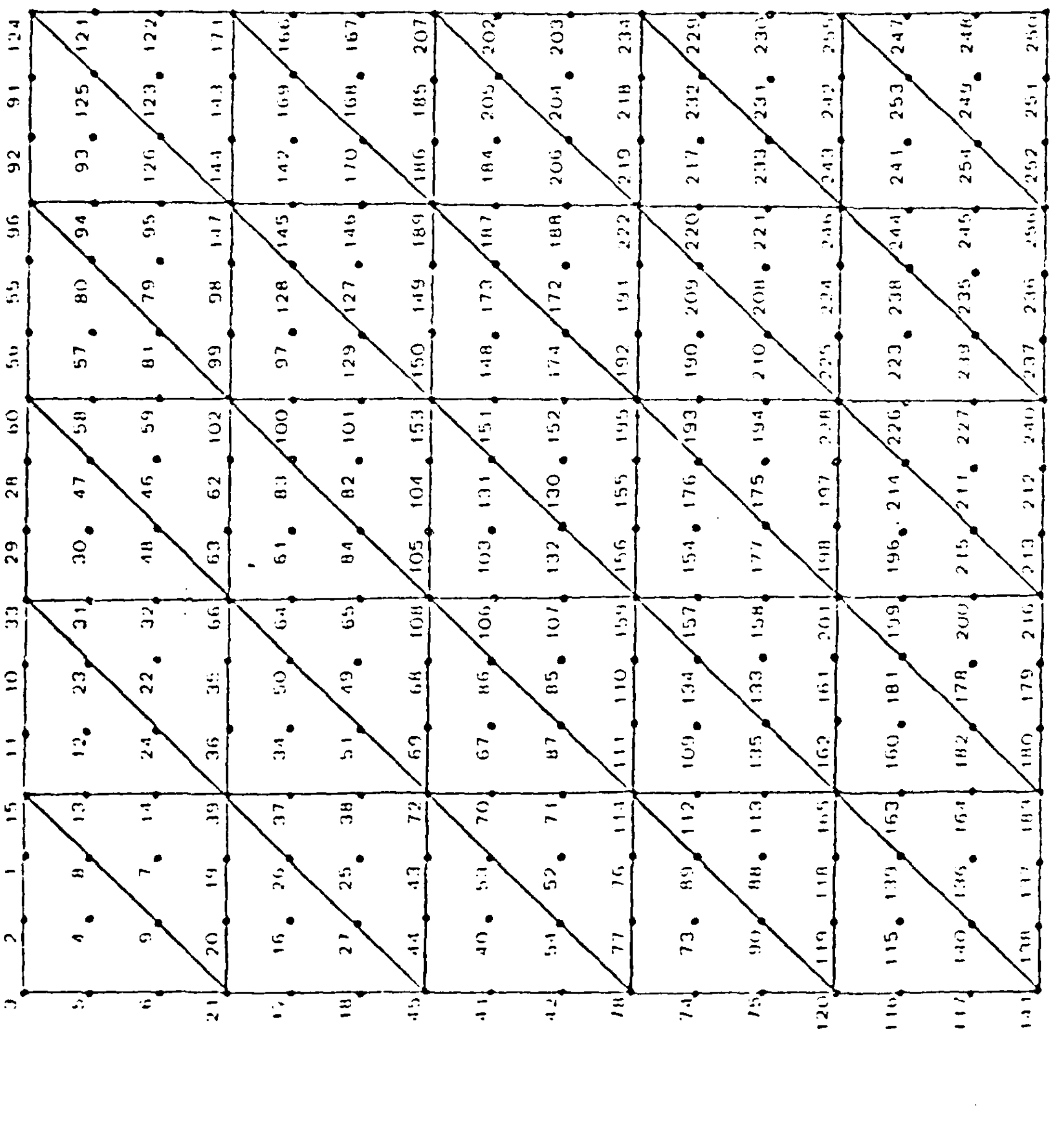
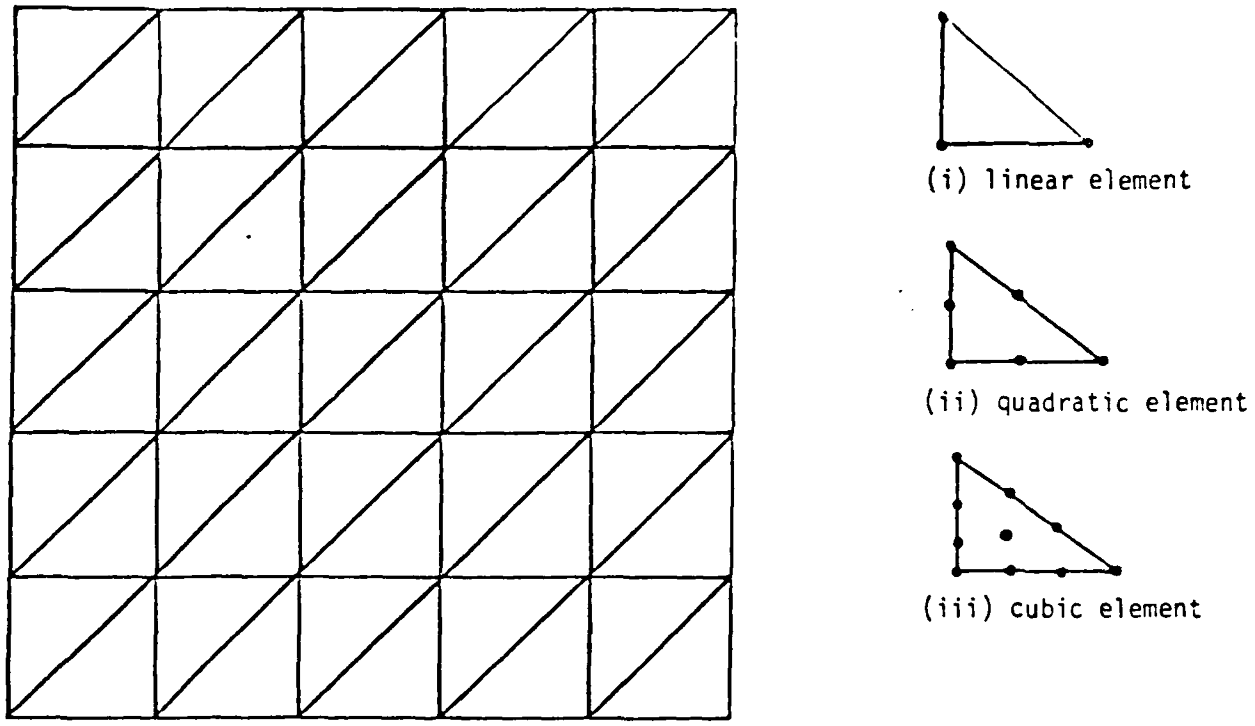
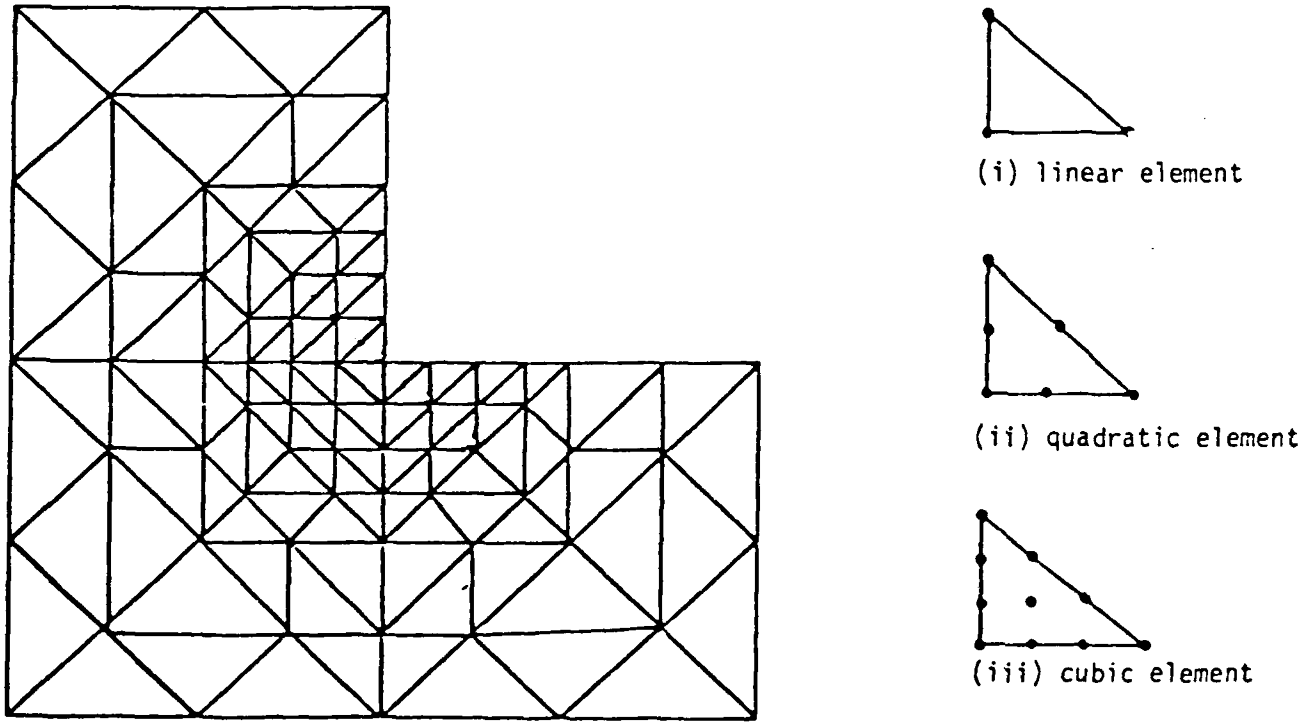


Figure 6: The Node Ordering of a 5 by 5 Regular Mesh with 10-Node Triangular Finite Elements

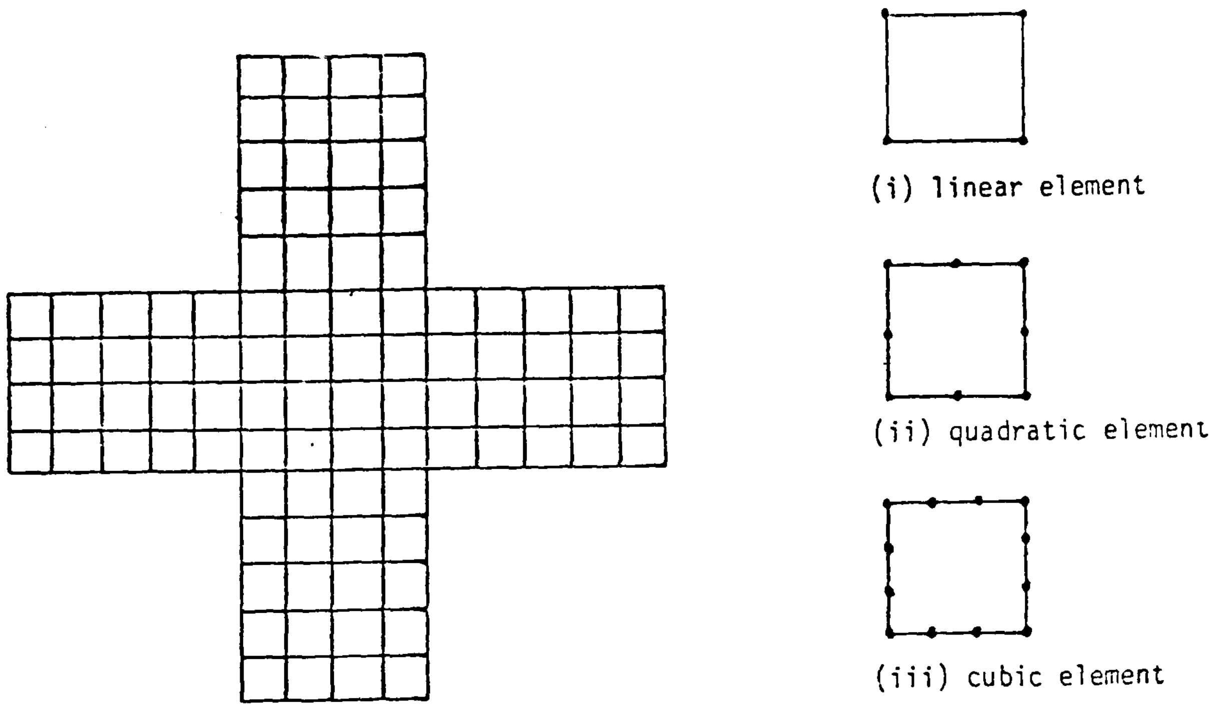


(a) 5 by 5 regular mesh with triangular finite elements

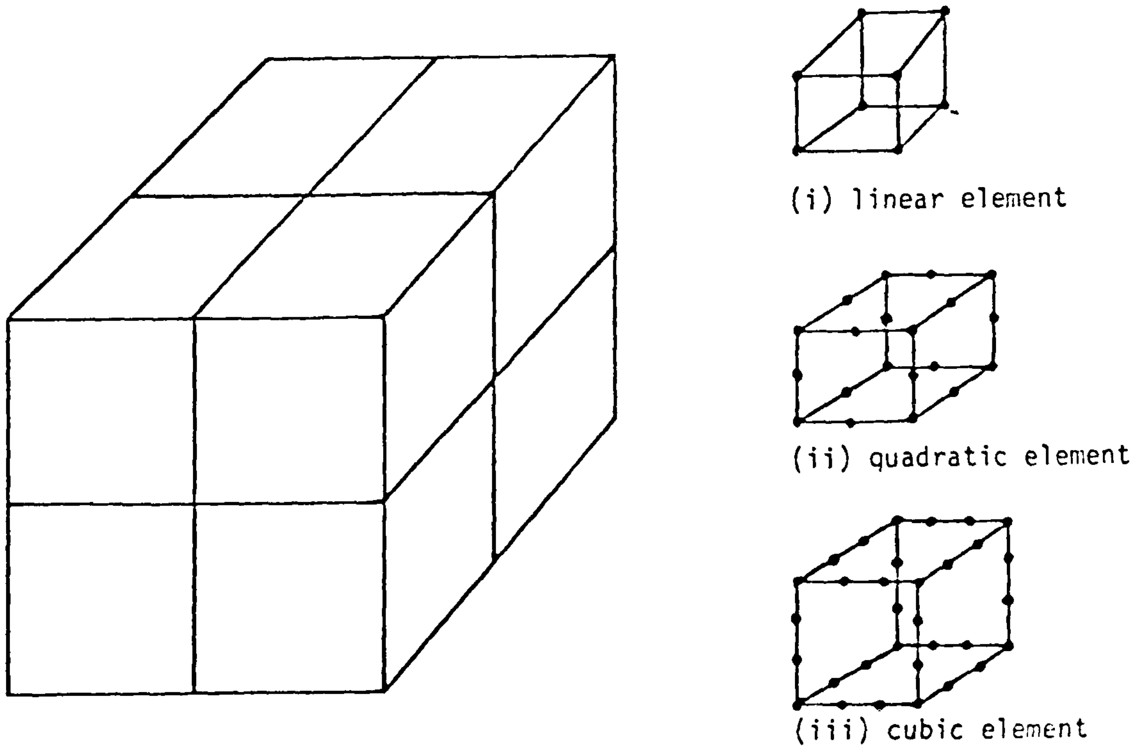


(b) L-shpaced mesh with triangular finite elements

Figure 7: Finite Element Models

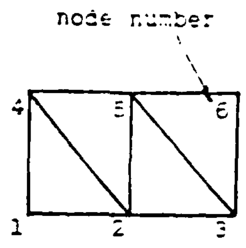


(c) Cross-shaped mesh with rectangular finite elements



(d) 2 by 2 by 2 block with volumetric finite elements

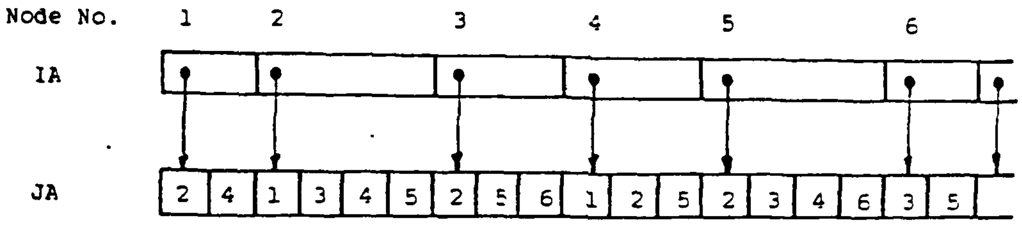
Figure 7: Finite Element Models (cont.)



(i) Finite element solution graph

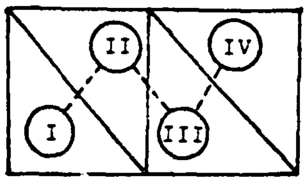
Node	Adj. nodes
1	2 4
2	1 3 4 5
3	2 5 6
4	1 2 5
5	2 3 4 6
6	3 5

(ii) Node connection table



(iii) Node-node adjacency structure pair (IA,JA)

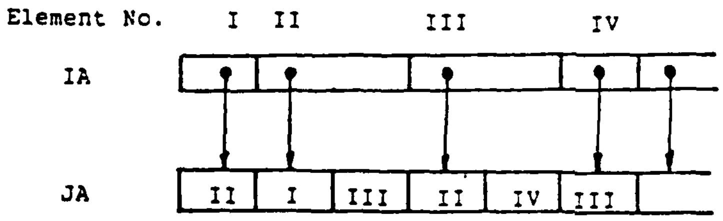
(a) Adjacency structure pair of finite element solution graph



(i) Finite element connectivity graph

Element	Adj. elements
I	II
II	I III
III	II IV
IV	III

(ii) Element connection table



(iii) Element-element adjacency structure pair (IA,JA)

(b) Adjacency structure pair of finite element connectivity graph

Figure 8: Examples of Adjacency Structure Pair

Table 1: Execution Time (msec) for RCM Algorithm and Two-Step Ordering Scheme

Model	No. of Ele-ments	Type of Ele-ment	No. of nodes per element	No. of Equa-tions	RCM algorithm (msec)	Two-step scheme (msec)			Two-Step / RCM
						Element ordering	Node Ordering	Total	
5 by 5 regular mesh	50	linear	3	36	12	10	3	13	1.08
		quad.	6	121	46	10	10	20	0.43
		cubic	10	256	145	10	17	27	0.19
L-shaped mesh	120	linear	3	73	23	29	14	43	1.87
		quad.	6	265	111	29	19	48	0.43
		cubic	10	577	373	29	41	70	0.19
Cross-shaped mesh	96	linear	4	96	34	18	11	29	0.85
		quad.	8	345	142	18	27	45	0.32
		cubic	12	565	319	18	40	58	0.18
2 by 2 by 2 block	8	linear	4	27	14	2	2	4	0.29
		quad.	20	81	81	2	8	10	0.12
		cubic	32	135	192	2	10	12	0.06

Table 2: Structure of Matrix Factor L Resulting from the RCM Algorithm and the Two-Step Ordering Scheme

Model	No. of Ele-ments	Type of Ele-ment	No. of nodes per element	No. of Equa-tions	No. of nonzeros in Δ of K	RCM algorithm			Two-step scheme			Ratio: Two-step/RCM		
						Profile	No. of fill-in	$\lvert L \rvert$	Profile	No. of fill-in	$\lvert L \rvert$	Profile	No. of fill-in	$\lvert L \rvert$
5 by 5 regular mesh	50	linear	3	36	85	155	70	155	155	70	155	1.00	1.00	1.00
		quad.	6	121	555	1189	634	1189	1235	576	1131	1.04	0.91	0.95
		cubic	10	256	1860	4140	2280	4140	4110	1458	3318	0.99	0.64	0.80
L-shaped mesh	120	linear	3	73	192	510	318	510	499	307	499	0.98	0.97	0.98
		quad.	6	265	1296	4208	2912	4208	4458	2864	4160	1.06	0.98	0.99
		cubic	10	577	4392	14496	10104	14496	15111	7368	11760	1.04	0.73	0.81
Cross-shaped mesh	96	linear	4	96	412	1298	669	1081	1702	646	1058	1.31	0.97	0.98
		quad.	8	345	2196	7282	3408	5604	9318	3222	5418	1.28	0.95	0.97
		cubic	12	565	5352	17832	8457	13809	22788	7728	13080	1.28	0.91	0.95
2 by 2 by 2 block	8	linear	4	27	158	230	72	230	230	42	200	1.00	0.58	0.87
		quad.	20	81	1202	1823	621	1823	1823	309	1511	1.00	0.50	0.83
		cubic	32	135	3212	4916	1704	4916	4916	822	3212	1.00	0.48	0.65

Appendix I. Fortran Computer Program for Labelling the Nodal Variables

Given the ordering of the finite elements, the subroutine listed below labels the nodal variables in the finite element mesh.

```
C/**********************************************************************
C/*                                                                    *
C/*     Subroutine NODORD :                                            *
C/*       Given the ordering of the finite elements, ELPM,             *
C/*             this subroutine NODORD labels the nodal variables      *
C/*             as follows                                             *
C/*       1. Determine the valency of each node, given the             *
C/*             element-node incidence table INC                       *
C/*       2. Label the nodes of each finite element, in decreasing     *
C/*             order of their valencies (linear insertion is used)    *
C/*       3. Reverse the node ordering obtained from step 2.           *
C/*                                                                    *
C/*     Input Variables :                                              *
C/*       INC - node-element INCidence table                           *
C/*       ELPM - ELement PerMutation vector                            *
C/*       NV - Number of Vertices in the finite element mesh           *
C/*       NVE - Number of Vertices per finite Element                  *
C/*       NE - Number of finite Elements in the mesh                   *
C/*                                                                    *
C/*     Output Variables :                                             *
C/*       PERM - node PERMutation vector                               *
C/*                                                                    *
C/*     Working Variables :                                            *
C/*       INVP - INVrse Permutation vector                             *
C/*       NDVL - NoDe VaLency vector                                   *
C/*                                                                    *
C/**********************************************************************
        SUBROUTINE NODORD(ELPM, INC, PERM, INVP, NDVL, NV, NVE, NE)
        IMPLICIT INTEGER (A-Z)
        DIMENSION INC(NE,NVE), PERM(NV), INVP(NV), ELPM(NE), NDVL(NV)
C/*
C/*     Initialization
C/*
        DO 10 K = 1,NV
          PERM(K) = 0
          INVP(K) = 0
          NDVL(K) = 0
   10   CONTINUE
C/*
C/*     Determine node valency
C/*
        DO 20 I = 1,NE
        DO 20 K = 1,NVE
          NODE = INC(I,K)
          IF(NODE.EQ.0) GOTO 20
          NDVL(NODE) = NDVL(NODE) + 1
   20   CONTINUE
C/*
C/*     Enter the elements sequentially following their orderings
C/*
        NXTNOD = 0
        DO 150 I = 1,NE
          ELEM = ELPM(I)
          START = NXTNOD + 1
          DO 110 K = 1,NVE
            NODE = INC(ELEM, K)
            IF(INVP(NODE).NE.0.OR.NODE.EQ.0) GOTO 110
              NXTNOD = NXTNOD + 1
              PERM(NXTNOD) = NODE
              INVP(NODE) = NXTNOD
  110     CONTINUE
C/*
C/*     Local node ordering : nodes  are sorted in decreasing
C/*             order of their valencies using linear insertion
C/*
          IF(START .GE. NXTNOD) GOTO 150
          K = START
  120     L = K
            K = K + 1
            PERMK = PERM(K)
            NDVLK = NDVL(PERMK)
  130       IF(L .LT. START) GOTO 140
                PERML = PERM(L)
                NDVLL = NDVL(PERML)
                IF(NDVLL .GE. NDVLK) GOTO 140
                  PERM(L+1) = PERML
                  L = L - 1
              GOTO 130
  140       PERM(L+1) = PERMK
            IF(K .LT. NXTNOD) GOTO 120
  150   CONTINUE
C/*
C/*     Now, reverse the node ordering
C/*
        KMID = NV/2
        J = NV
        DO 160 K = 1,KMID
          I = PERM(K)
          PERM(K) = PERM(J)
          PERM(J) = I
          J = J - 1
  160   CONTINUE
        RETURN
        END
```

Appendix II. Fortran Computer Program for Generating the Adjacency Structure Pair of Element-Element Connectivity

Given the element-node incidence table, the following set of subroutines determines the adjacency structure pair of element-element connectivity of an arbitrary finite element mesh. These subroutines are included only for illustrating how the element-element adjacency may be generated if the mesh generator produces only the element-node incidence table. The subroutines are not efficient in storage, in that space is provided for the entire symmetric node-node adjacency table, denoted by NEL. Since this table is very sparse, the function HASH could be replaced by one that exploits sparsity[17] (and the variable NEDGE reduced accordingly), or other efficient techniques for storing sparse tables may be used[22].

```
C/*********************************************************************
C/*                                                                   *
C/*  Subroutine SETUP                                                 *
C/*     Purpose : SETUP is the main subroutine to generate the        *
C/*               element-element adjacency structure from the        *
C/*               element-node incidence table, INC, for a planar     *
C/*               finite element mesh.                                *
C/*                                                                   *
C/*     Input Variables :                                             *
C/*       INC - element-node incidence table                          *
C/*                                                                   *
C/*     Global Variables :                                            *
C/*       NE - no. of elements in the mesh                            *
C/*       NV - no. of nodes in the mesh                               *
C/*       NVE - no. of nodes per element                              *
C/*       NEPLS1 - (NE + 1)                                           *
C/*       NEDGE - possible maximum no. of edges in the solution graph *
C/*               ( = NV * (NV-1) / 2   )                             *
C/*       EDGMAX - total no. of edges in the solution graph           *
C/*       EDGPL1 - (EDGMAX + 1)                                       *
C/*       NELS - counter for the edge-element structure pair          *
C/*       MAXSUB - max. subscript used in the element-element         *
C/*                adjacency structure pair (see subroutine ELMADJ)   *
C/*                                                                   *
C/*     Output :                                                      *
C/*       The element-element adjacent structure pair (IA,JA) is stored *
C/*     in the array POOL from location LOCIA to (LOCJA+MAXSUB-1).    *
C/*     The array IA is stored in POOL(LOCIA) to POOL(LOCIA+NE);      *
C/*     the array JA is stored in POOL(LOCJA) to POOL(LOCJA+MAXSUB-1). *
C/*                                                                   *
C/*     Subroutines used :                                            *
C/*       EDGSET, EDGBLT, ELMADJ                                      *
C/*                                                                   *
C/*********************************************************************
      SUBROUTINE SETUP(INC, POOL)
      IMPLICIT INTEGER (A-Z)
      COMMON /SYSTEM/ NE, NV, NVE, NEPLS1
      COMMON /GRAPH/ NEDGE, EDGMAX, EDGPL1, NELS, MAXSUB
      COMMON /IOSET/ IN, IO, DEBUG
      LOGICAL DEBUG
      DIMENSION INC(NE,NVE), POOL(1)
C/*
C/*       Phase  1 : to set up the edge-element adjacency structure
C/*
      LOCNEL = 1
      LOCSTA = LOCNEL + NEDGE
      CALL EDGSET(INC, POOL(LOCNEL), POOL(LOCSTA))
C/*
C/*       Phase  2 : to build the edge-element adjacency
C/*                       structure pair
C/*
      EDGPL1 = EDGMAX + 1
      LOCEL = LOCSTA + EDGPL1
      LOCPOS = LOCEL + NELS
      CALL EDGBLT(INC, POOL(LOCNEL), POOL(LOCSTA), POOL(LOCEL),
     *         POOL(LOCPOS))
C/*
C/*       Phase  3 : to build the element-element adjacency
C/*                       structure pair
C/*
      LOCIA = LOCPOS
      LOCJA = LOCIA + NEPLS1
      CALL ELMADJ(INC, POOL(LOCNEL), POOL(LOCSTA), POCL(LOCEL),
     *         POOL(LOCIA), POOL(LOCJA))
      RETURN
      END
C/*********************************************************************
C/*                                                                  *
C/*   Function HASH                                                  *
C/*     Purpose : to determine the location of an edge in            *
C/*             the symmetric node adjacency matrix                  *
C/*                                                                  *
C/*     Input Variables :                                            *
C/*       N1, N2 - nodes in the mesh (or graph)                      *
C/*                                                                  *
C/*     Output Variable :                                            *
C/*       HASH - location of edge (N1,N2) in the symmetric node      *
C/*             adjacency matrix                                     *
C/*                                                                  *
C/*********************************************************************
      INTEGER FUNCTION HASH(N1,N2)
      IMPLICIT INTEGER (A-Z)
      J = AMAXO(N1,N2)
      I = AMINO(N1,N2)
      HASH = ((J-2)*(J-1))/2 + I
      RETURN
      END
C/*********************************************************************
C/*                                                                  *
C/*   Subroutine EDGSET                                              *
C/*     Purpose : to set up counter START of the edge-element        *
C/*             adjacency structure pair (START, EL)                 *
C/*             {EL will be determined in subroutine EDGBLT.}        *
C/*                                                                  *
C/*     Input Variable :                                             *
```

```
C/*       INC - element-node incidence table                               *
C/*                                                                        *
C/*     Output Variables :                                                 *
C/*       NEL(HASH(N1,N2)) - edge no. assigned to node pair (N1,N2)        *
C/*             { Only edges with more than one incident element           *
C/*               are assigned a number. }                                 *
C/*       START - starting position for the edge-element adjacency         *
C/*             structure pair                                             *
C/*             { START(k+1) - START(k) = no. of elements adjacent         *
C/*                     to edge k }                                        *
C/*                                                                        *
C/*     Intermediate Variable :                                            *
C/*       NEL(HASH(N1,N2)) - in step 1, the array is a node-adjacency      *
C/*             matrix denoting number of elements into edge (N1,N2)       *
C/*                                                                        *
C/************************************************************************
        SUBROUTINE EDGSET(INC, NEL, START)
        IMPLICIT INTEGER (A-Z)
        COMMON /SYSTEM/ NE, NV, NVE, NEPLS1
        COMMON /GRAPH/ NEDGE, EDGMAX, EDGPL1, NELS, MAXSUB
        DIMENSION INC(NE,NVE), NEL(NEDGE), START(1)
C/*
C/*        Initialization
C/*
        DO 5 I = 1,NEDGE
          NEL(I) = 0
    5   CONTINUE
C/*
C/*        Step  1 : to build the node adjacency matrix
C/*
        DO 10 I = 1,NE
          NN = NVE
          DO 10 J = 1, NN-1
            N1 = INC(I,J)
            IF(N1.EQ.0) GOTO 10
            DO 10 K = J+1,NN
              N2 = INC(I,K)
              IF(N1.EQ.N2 .OR. N2.EQ.0) GOTO 10
              EDGE = HASH(N1,N2)
              NEL(EDGE) = NEL(EDGE) + 1
   10   CONTINUE
C/*
C/*        Step  2 :
C/*          (1) to set up counter START for the
C/*              edge-element adjacency structure
C/*          (2) to assign a number to the edges (with two
C/*              or more incident elements ).
C/*
        POSIT = 0
        NELS = 1
        DO 15 I = 1, NEDGE
          IF(NEL(I) .LE. 1) GOTO 14
            POSIT = POSIT + 1
            START(POSIT) = NELS
            NELS = NELS + NEL(I)
            NEL(I) = POSIT
          GOTO 15
   14       NEL(I) = 0
   15   CONTINUE
        EDGMAX = POSIT
        START(EDGMAX+1) = NELS
        RETURN
        END
C/********************************************************************
C/*                                                                  *
C/*   Subroutine EDGBLT                                              *
C/*     Purpose : to build the array EL of the edge-element adjacency  *
C/*             structure pair (START, EL).                          *
C/*                                                                  *
C/*     Input Variables :                                            *
C/*       INC - element-node incidence table                         *
C/*       NEL(k) - edge no. assigned to node pair (N1,N2)            *
C/*             { k = HASH(N1,N2) }                                  *
C/*       START - starting position for the edge-element adjacency   *
C/*             structure pair                                       *
C/*             { START(k+1) - START(k) = no. of elements adjacent   *
C/*                     to edge k }                                  *
C/*                                                                  *
C/*     Output Variable :                                            *
C/*       EL -  edge-element adjacency structure pair                *
C/*             { The set of elements incident to edge k is stored   *
C/*               in EL(START(k)) to EL(START(k+1)). }               *
C/*                                                                  *
C/*     Working Variable :                                           *
C/*       POSIT(k) - pointer to current empty position in EL for edge k *
C/*                                                                  *
C/********************************************************************
        SUBROUTINE EDGBLT(INC, NEL, START, EL, POSIT)
        IMPLICIT INTEGER (A-Z)
        COMMON /SYSTEM/ NE, NV, NVE, NEPLS1
        COMMON /GRAPH/ NEDGE, EDGMAX, EDGPL1, NELS, MAXSUB
        DIMENSION INC(NE,NVE), NEL(NEDGE), START(EDGPL1), EL(NELS),
     *          POSIT(EDGMAX)
C/*
C/*       Initialize working vector POSIT
C/*
        DO 10 I = 1, EDGMAX
          POSIT(I) = START(I)
   10   CONTINUE
C/*
C/*       Fill edge-element adjacency EL
C/*
        DO 20 I = 1,NE
          NN = NVE
          DO 20 J = 1, NN-1
            N1 = INC(I,J)                                !node N1!
            IF(N1.EQ.0) GOTO 20
            DO 20 K = J+1, NN
              N2 = INC(I,K)                              !node N2!
              IF(N2.EQ.0 .OR. N1.EQ.N2) GOTO 20          !check self loop!
                EDGE = HASH(N1,N2)                       !location in NEL!
                IF(NEL(EDGE) .EQ. 0) GOTO 20
                EDGACT = NEL(EDGE)                       !edge number!
                EL(POSIT(EDGACT)) = I                    !assign I to EL!
                POSIT(EDGACT) = POSIT(EDGACT) + 1        !update POSITion!
   20   CONTINUE
```

```
        RETURN
        END
C/************************************************************************
C/*                                                                      *
C/*   Subroutine ELMADJ                                                  *
C/*     Purpose : to set up the element-element adjacency structure      *
C/*            pair (IADJ,JADJ).                                         *
C/*                                                                      *
C/*     Input Variables :                                                *
C/*       INC - element-node incidence table                             *
C/*       NEL(k) - edge no. assigned to node pair (N1,N2)                *
C/*             { k = HASH(N1,N2) }                                      *
C/*       START - starting position for the edge-element adjacency       *
C/*             structure pair                                           *
C/*             { START(k+1) - START(k) = no. of elements adjacent       *
C/*                     to edge k }                                      *
C/*       EL -  edge-element adjacency structure pair                    *
C/*             { The set of elements incident to edge k is stored       *
C/*               in EL(START(k)) to EL(START(k+1)). }                   *
C/*                                                                      *
C/*     Output Variables :                                               *
C/*       (IADJ,JADJ) - element-element adjacency structure pair         *
C/*                                                                      *
C/************************************************************************
        SUBROUTINE ELMADJ(INC, NEL, START, EL, IADJ, JADJ)
        IMPLICIT INTEGER (A-Z)
        COMMON /SYSTEM/ NE, NV, NVE, NEPLS1
        COMMON /GRAPH/ NEDGE, EDGMAX, EDGPL1, NELS, MAXSUB
        COMMON /IOSET/ IN, IO, DEBUG
        LOGICAL DEBUG
        DIMENSION INC(NE,NVE), NEL(NEDGE), START(EDGPL1), EL(NELS),
     *          IADJ(NEPLS1), JADJ(1)
C/*
C/*       Initialize IADJ(1)
C/*
        IADJ(1) = 1
C/*
C/*       For each element, find all incident elements and
C/*            queue them in JADJ
C/*
        DO 50 I = 1,NE
          ELMENT = I                                   !for element I!
          L1 = IADJ(I)
          FILL = 0
          NN = NVE
          DO 40 J = 1, NN-1
            N1 = INC(I,J)                              !node N1!
            IF(N1.EQ.0) GOTO 40
            DO 40 K = J+1, NN
              N2 = INC(I,K)                            !node N2!
              IF(N1.EQ.N2 .OR. N2.EQ.0) GOTO 40        !check self loop!
                EDGE = HASH(N1,N2)                     !location in NEL!
                IF(NEL(EDGE).EQ.0) GOTO 40
                EDGNUM = NEL(EDGE)                     !edge number!
                BEGIN = START(EDGNUM)                  !for all elements
                END = START(EDGNUM+1) - 1              ! incident to
                DO 30 IEL = BEGIN, END                 ! edge EDGNUM!
                  ELM = EL(IEL)                        !a neigh. element!
                IF(ELM.EQ.ELMENT) GOTO 30             !same as element I?!
                  L2 = L1 + FILL
                  IF(FILL.EQ.0) GOTO 20
                    DO 10 JJ = L1, L2-1               !check if there
                      IF(JADJ(JJ).EQ.ELM) GOTO 30     ! is a duplication
   10               CONTINUE                          ! in list        !
   20             JADJ(L2) = ELM                      !queue it in JADJ!
                  FILL = FILL + 1
   30         CONTINUE
   40     CONTINUE
          IADJ(ELMENT+1) = IADJ(ELMENT) + FILL        !update IADJ!
   50   CONTINUE
        MAXSUB = IADJ(NEPLS1)
        IF(.NOT.DEBUG) RETURN
C/*
C/*        Print out results
C/*
        WRITE(IO,1100)
        DO 60 I = 1,NE
          L1 = IADJ(I)
          L2 = IADJ(I+1) - 1
          IF(L2.LT.L1) GOTO 60
            WRITE(IO,1000) I, L1, L2, (JADJ(K), K=L1,L2)
   60   CONTINUE
        RETURN
 1000   FORMAT(T5, I4, T10, I6, T18, I6, (T30,10I6))
 1100   FORMAT(1H1, T2, 'ELEMENT', T10, 'IDAJ(I)', T18, 'IADJ(I+1)',
     *          T30, 'JADJ (NEIGHBORING ELEMENTS) :',/)
        END
```

[illegible]

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER R-81-130 / 2. GOVT ACCESSION NO. AD-A104320	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A TWO-STEP APPROACH TO FINITE ELEMENT ORDERING	5. TYPE OF REPORT & PERIOD COVERED Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Steven J. Fenves, Kincho H. Law	8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0354 ✓
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Civil Engineering Carnegie-Mellon University Pittsburgh, Pennsylvania 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington VA 22217	12. REPORT DATE August 1981
	13. NUMBER OF PAGES 39
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Two-step ordering; finite element ordering; sparse matrices; graph-theoretic approach; solution graph; connectivity graph; Cuthill-McKee algorithm; RCM algorithm; execution time; fill-in entries.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A two-step approach to finite element ordering is introduced. The scheme involves ordering of the finite elements first, based on their adjacency, followed by a local numbering of the nodal variables. The ordering of the elements is performed by the Cuthill-McKee algorithm. This approach takes into consideration the underlying structure of the finite element mesh, and may be regarded as a "natural" finite element ordering scheme. The experimental results show that this two-step scheme is more efficient than the reverse Cuthill-McKee algorithm applied directly to the nodes, in terms

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20 Abstract (Continued)

of both execution time and the number of fill-in entries, particularly when higher order finite elements are used. In addition to its efficiency, the two-step approach increases modularity and flexibility in finite element programs, and possesses potential application to a number of finite element solution methods.

DATE
ILME